Fallout: Leaking Data on Meltdown-resistant CPUs

Claudio Canella[1], Daniel Genkin[2], Lukas Giner[1], Daniel Gruss[1], Moritz Lipp[1], Marina Minkin[2],

Daniel Moghimi[3], Frank Piessens[4], Michael Schwarz[1], Berk Sunar[3], Jo Van Bulck[4], Yuval Yarom[5]

[1]Graz University of Technology, [2]University of Michigan, [3]Worcester Polytechnic Institute,

[4]imec-DistriNet, KU Leuven, [5]The University of Adelaide and Data61

ABSTRACT

Meltdown and Spectre enable arbitrary data leakage from memory via various side channels. Short-term software mitigations for Meltdown are only a temporary solution with a significant performance overhead. Due to hardware fixes, these mitigations are disabled on recent processors.

In this paper, we show that Meltdown-like attacks are still possible on recent CPUs which are not vulnerable to Meltdown. We identify two behaviors of the store buffer, a microarchitectural resource to reduce the latency for data stores, that enable powerful attacks. The first behavior, *Write Transient Forwarding* forwards data from stores to subsequent loads even when the load address differs from that of the store. The second, *Store-to-Leak* exploits the interaction between the TLB and the store buffer to leak metadata on store addresses. Based on these, we develop multiple attacks and demonstrate data leakage, control flow recovery, and attacks on ASLR. Our paper shows that Meltdown-like attacks are still possible, and software fixes with potentially significant performance overheads are still necessary to ensure proper isolation between the kernel and user space.

KEYWORDS

side-channel attack, Meltdown, Spectre, store buffer, store-to-load

ACM Reference Format:
Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3363219

1 INTRODUCTION

The computer architecture and security communities will remember 2018 as the year of Spectre and Meltdown [47, 51]. Speculative and out-of-order execution, which have been considered for decades to be harmless and valuable performance features, were discovered to have dangerous industry-wide security implications, affecting operating systems [47, 51], browsers [1, 47], virtual machines [78], Intel SGX [74] and cryptographic hardware accelerators [72].

Recognizing the danger posed by this new class of attacks, the computer industry mobilized. For existing systems, all major OSs deployed the KAISER countermeasure [25], e.g., on Linux under the name KPTI, potentially incurring significant performance losses [23]. For newer systems, Intel announced a new generation of silicon-based countermeasures, mitigating many transient-execution attacks directly in hardware, while improving overall performance [15].

However, while Intel claims that these fixes correctly address the hardware issues behind Meltdown and Foreshadow, it remains unclear whether new generations of Intel processors are properly protected against Meltdown-type transient-execution attacks. Thus, in this work we set out to investigate the following questions:

Are new generations of processors adequately protected against transient-execution attacks? If so, can ad-hoc software mitigations be safely disabled on post-Meltdown Intel hardware?

Our Contributions. Unfortunately, this paper answers these questions in the negative, showing that data leakage is still possible even on newer Meltdown-protected Intel hardware, which avoids the use of older software countermeasures. At the microarchitectural level, in this work, we focus on the store buffer, a microarchitectural element which serializes the stream of stores and hides the latency of storing values to memory. In addition to showing how to effectively leak the contents of this buffer to read kernel writes from user space, we also contribute a novel side channel on the translation lookaside buffer (TLB), named Store-to-Leak, that exploits the lacking permission checks within Intel's implementation of the store buffer to break KASLR, to break ASLR from JavaScript, and to infer the kernel control flow.

Thus, in this work we make the following contributions:

(1) We discover a security flaw due to a shortcut in Intel CPUs, which we call *Write Transient Forwarding* (WTF), that allows us to read the data corresponding to recent writes.
(2) We demonstrate the security implications of the WTF shortcut by recovering the values of recent writes performed by the OS kernel, recovering data from within TSX transactions, as well as leaking cryptographic keys.
(3) We identify a new TLB side channel, which we call *Store-to-Leak*. Store-to-Leak exploits Intel's store-to-load forwarding unit in order to reveal when an inaccessible virtual store address is mapped to a corresponding physical store address by

CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363219

exploiting a missing permission check when forwarding from these addresses.

(4) We demonstrate how to exploit Store-to-Leak for breaking KASLR and ASLR from JavaScript, as well as how to use it to simplify the gadget requirements for Spectre-style attacks.

(5) We identify a new cause for transient execution, namely *assists*, which are small microcode routines that execute when the processor encounters specific edge-case conditions.

(6) We implement the first documented Meltdown-type attacks that exploit page fault exceptions due to Supervisor Mode Access Prevention (SMAP).

Responsible Disclosure. Store-to-leak was responsibly disclosed to Intel by the authors from Graz University of Technology on January 18, 2019. Write Transient Forwarding was then responsibly disclosed to Intel by the authors from the University of Michigan, and University of Adelaide and Data61, on January 31, 2019. Intel indicated that it was previously aware of the Write Transient Forwarding issue, assigning it CVE-2018-12126, Microarchitectural Store Buffer Data Sampling (MSBDS). According to Intel, we were the first academic groups to report the two respective issues.

Write Transient Forwarding was also disclosed to AMD, ARM, and IBM, which stated that none of their CPUs are affected.

RIDL and ZombieLoad. In concurrent works, RIDL [76] and ZombieLoad [68] demonstrate leakage from the Line Fill Buffer (LFB) and load ports on Intel processors. They show that faulty loads can also leak data from these other microarchitectural resources across various security domains. Fallout is different from and complementary to the aforementioned contributions, as it attacks the store buffer and store instructions, as opposed to loads. RIDL, ZombieLoad, and this work were disclosed to the public under the umbrella name of *Microarchitectural Data Sampling* (MDS).

2 BACKGROUND

In this section, we present background regarding cache attacks, transient-execution attacks, Intel's store buffer implementation, virtual-to-physical address translation, and finally address-space-layout randomization (ASLR).

2.1 Cache Attacks

Processor speed increased by several orders of magnitude over the past decades. While the bandwidth of modern main memory (DRAM) has also increased, the latency has not kept up with the change. Consequently, the processor needs to fetch data from DRAM ahead of time and buffer it in faster internal storage. For this purpose, processors contain small memory buffers, called caches, that store frequently or recently accessed data. In modern processors, the cache is organized in a hierarchy of multiple levels, with the lowest level being the smallest but also the fastest.

Caches are used to hide the latency of memory accesses, as there is a speed gap between the processor and DRAM. As a result, caches inherently introduce timing channels. A multitude of cache attacks have been proposed over the past two decades [7, 28, 62, 80]. Today, the most important practical attack techniques are Prime+Probe [62, 63] and Flush+Reload [80]. Some of these techniques exploit the last-level cache, which is shared and inclusive on most processors. Prime+Probe attacks constantly measure how long it takes to fill an entire cache set. Whenever a victim process accesses a cache line in this cache set, the measured time will be slightly higher. In a Flush+Reload attack, the attacker constantly flushes the targeted memory location, e.g., using the `clflush` instruction. The attacker then measures how long it takes to reload the data. Based on the reload time, the attacker determines whether a victim has accessed the data in the meantime. Flush+Reload has been used for attacks on various computations, e.g., web server function calls [81], user input [29, 50, 67], kernel addressing information [27], and cryptographic algorithms [6, 8, 19, 43, 64, 80].

Covert channels represent a slightly different scenario, in which the attacker, who controls both the sender and receiver, aims to circumvent the security policy, leaking information between security domains. Both Flush+Reload and Prime+Probe have been used as high-performance covert channels [28, 52, 56].

2.2 Transient-Execution Attacks

Program code can be represented as a stream of instructions. Following this instruction stream in strict order would result in numerous processor stalls while instructions wait for all operands to become available, even though subsequent instructions may be ready to run. To optimize this case, modern processors first fetch and decode instructions in the front end. In many cases, instructions are split up into smaller micro-operations (μOPs) [18]. These μOPs are then placed in the so-called Reorder Buffer (ROB). μOPs that have operands also need storage space for these operands. When a μOP is placed in the ROB, this storage space is dynamically allocated from the load buffer for memory loads, the store buffer for memory stores, and the register file for register operations. The ROB entry only references the load and store buffer entries. While the operands of a μOP still may not be available when it is placed in the ROB, the processor can now schedule subsequent μOPs. When a μOP is ready to be executed, the scheduler schedules it for execution. The results of the execution are placed in the corresponding registers, load buffer entries, or store buffer entries. When the next μOP in order is marked as finished, it is retired, and the buffered results are committed and become architectural.

As software is rarely purely linear, the processor has to either stall execution until a (conditional) branch is resolved or speculate on the most likely outcome and start executing along the predicted path. The results of those predicted instructions are placed in the ROB until the prediction is verified. If the prediction was correct, the predicted instructions are retired in order. Otherwise, the processor flushes the pipeline and the ROB without committing any architectural changes and execution continues along the correct path. However, microarchitectural state changes, such as loading data into the cache or TLB, are not reverted. Similarly, when an interrupt occurs, operations already executed out of order must be flushed from the ROB. We refer to instructions that have been executed but never committed as *transient instructions* [10, 47, 51]. Spectre-type attacks [10, 11, 35, 46–48, 54] exploit the transient execution of instructions before a misprediction is detected. Meltdown-type attacks [5, 10, 39, 40, 46, 51, 72, 74, 78] exploit the transient execution of instructions before a fault is handled.

EXHIBIT 3

2.3 Store Buffer

When the execution unit needs to write data to memory, instead of waiting for the completion of the store, it merely enqueues the request in the store buffer. This allows the CPU to continue executing instructions from the current execution stream, without having to wait for the write to finish. This optimization makes sense, as in many cases writes do not influence subsequent instructions, i.e., only loads to the same address should be affected. Meanwhile, the store buffer asynchronously processes the stores, ensuring that the results are written to memory. Thus, the store buffer prevents CPU stalls while waiting for the memory subsystem to finish the write. At the same time, it guarantees that writes reach the memory subsystem in order, despite out-of-order execution.

For every store operation that is added to the ROB, the CPU allocates an entry in the store buffer. This entry requires both the virtual and physical address of the target. On Intel CPUs, the store buffer has up to 56 entries [42], allowing for up to 56 stores to be handled concurrently. Only if the store buffer is full, the front end stalls until an empty slot becomes available again [42].

Although the store buffer hides the latency of stores, it also increases the complexity of loads. Every load has to search the store buffer for pending stores to the same address in parallel to the regular L1 lookup. If the full address of a load matches the full address of a preceding store, the value from the store buffer entry can be used directly. This optimization for subsequent loads is called *store-to-load forwarding* [34].

2.4 Address Translation and TLB

Memory isolation is the basis of modern operating system security. For this purpose, processes operate on virtual instead of physical addresses and are architecturally prevented from interfering with each other. The processor translates virtual to physical addresses through a multi-level page-translation table. The process-specific base address of the top-level table is kept in a dedicated register, e.g., CR3 on x86, which is updated upon a context switch. The page table entries track various properties of the virtual memory region, e.g., user-accessible, read-only, non-executable, and present.

The translation of a virtual to a physical address is time-consuming. Therefore, processors have special caches, translation-lookaside buffers (TLBs), which cache page table entries [38].

2.5 Address Space Layout Randomization

To exploit a memory corruption bug, an attacker often requires knowledge of addresses of specific data. To impede such attacks, different techniques like address space layout randomization (ASLR), non-executable stacks, and stack canaries have been developed. KASLR extends ASLR to the kernel, randomizing the offsets where code, data, drivers, and other mappings are located on every boot. The attacker then has to guess the location of (kernel) data structures, making attacks harder.

The double page fault attack by Hund et al. [36] breaks KASLR. An unprivileged attacker accesses a kernel memory location and triggers a page fault. The operating system handles the page fault interrupt and hands control back to an error handler in the user program. The attacker now measures how much time passed since triggering the page fault. Even though the kernel address is inaccessible to the user, the address translation entries are copied into the TLB. The attacker now repeats the attack steps, measuring the execution time of a second page fault to the same address. If the memory location is valid, the handling of the second page fault will take less time as the translation is cached in the TLB. Thus, the attacker learns whether a memory location is valid even though the address is inaccessible to user space.

The same effect has been exploited by Jang et al. [45] in combination with Intel TSX. Intel TSX extends the x86 instruction set with support for hardware transactional memory via so-called TSX transactions. A TSX transaction is aborted without any operating system interaction if a page fault occurs within it. This reduces the noise in the timing differences that was present in the attack by Hund et al. [36] as the page fault handling of the operating system is skipped. Thus, the attacker learns whether a kernel memory location is valid with almost no noise at all.

The prefetch side channel presented by Gruss et al. [27] exploits the software prefetch instruction. The execution time of the instruction is dependent on the translation cache that holds the correct entry. Thus, the attacker not only learns whether an inaccessible address is valid but also the corresponding page size.

3 ATTACK PRIMITIVES

In this section, we introduce the underlying mechanisms for the attacks we present in this paper. First, we introduce the Write Transient Forwarding (WTF) shortcut, that allows user applications to read kernel and TSX writes. We then describe three primitives based on Store-to-Leak, a side-channel that exploits the interaction between the store buffer and the TLB to leak information on the mapping of virtual addresses. We begin with *Data Bounce*, which exploits the conditions for Store-to-Leak to attack both user and kernel space ASLR (cf. Section 6). We then exploit interactions between Data Bounce and the TLB in the *Fetch+Bounce* primitive. Fetch+Bounce enables attacks on the kernel at a page-level granularity, similar to previous attacks [21, 24, 65, 79] (cf. Section 7). We conclude this section by augmenting Fetch+Bounce with speculative execution in *Speculative Fetch+Bounce*. Speculative Fetch+Bounce leads to usability improvement in Spectre attacks (cf. Section 8).

3.1 Write Transient Forwarding

In this section, we discuss the WTF shortcut, which incorrectly passes values from memory writes to subsequent faulting load instructions. More specifically, as explained in Section 2.3, when a program attempts to read from an address, the CPU must first check the store buffer for writes to the same address, and perform store-to-load forwarding if the addresses match. An algorithm for handling partial address matches appears in an Intel patent [33]. Remarkably, the patent explicitly states that:

> "if there is a hit at operation 302 [lower address match] and the physical address of the load or store operations is not valid, the physical address check at operation 310 [full physical address match] may be considered as a hit and the method 300 [store-to-load forwarding] may continue at operation 308 [block load/forward data from store]."

```
 1 char* victim_page = mmap(..., PAGE_SIZE, PROT_READ | PROT_WRITE,
 2                                          MAP_POPULATE, ...);
 3 char* attacker_address = 0x9876543214321000ull;
 4
 5 int offset = 7;
 6 victim_page[offset] = 42;
 7
 8 if (tsx_begin() == 0) {
 9   memory_access(lut + 4096 * attacker_address[offset]);
10   tsx_end();
11 }
12
13 for (i = 0; i < 256; i++) {
14   if (flush_reload(lut + i * 4096)) {
15     report(i);
16   }
17 }
```

Listing 1: Exploiting the WTF shortcut in a toy example.

That is if address translation of a load μOP fails and some lower address bits of the load match those of a prior store, the processor assumes that the physical addresses of the load and the store match and forwards the previously stored value to the load μOP. We note that the faulting load is transient and will not retire, hence WTF has no architectural implications. However, as we demonstrate in Section 4, the microarchitectural side effects of transient execution following the faulting load may result in inadvertent information leaks.

A Toy Example. We begin our investigation of the WTF shortcut with the toy example in Listing 1, which shows a short code snippet that exploits the WTF shortcut to read memory addresses without directly accessing them. While Listing 1 uses non-canonical addresses (i.e., a virtual address in which bits 47 to 63 are neither all '0' nor all '1') to cause a fault, other exception causes are also possible. We refer to Section 5.2 for a systematic analysis of different exception types that may trigger WTF. We choose non-canonical addresses for our first example, as these work reliably across all processor generations while imposing minimal constraints on the attacker.

Setup. Listing 1 begins by allocating a `victim_page`, which is a 'normal' page where the user can write and read data. It then defines the `attacker_address` variable, which points to a non-canonical address. Note that dereferencing such a pointer results in a general protection fault (#GP) [38], faulting the dereferencing load. We then store the secret value 42 to the specified offset 7 in the user-space accessible `victim_page`. This prompts the processor to allocate a store buffer entry for holding the secret value to be written out to the memory hierarchy.

Reading Previous Stores. We observe that the code in Listing 1 never reads from the `victim_page` directly. Instead, the attacker reads out the store buffer entry by dereferencing a distinct `attacker_address`. We suppress the general protection fault that results from this access using a TSX transaction (Line 8). Alternatively, the exception can be suppressed through speculative execution using a mispredicted branch [47], call [47], or return [48, 54]. However, the reorder buffer only handles the exception when the memory access operation retires. In the meantime, due to the WTF shortcut, the CPU transiently forwards the value of the previous store at the same page offset. Thus, the memory access picks-up the value of the store to `victim_page`, in this example the secret value 42. Using a cache-based covert channel, we transmit the incorrectly forwarded value (Line 9). Finally, when the failure and transaction abort are handled, no architectural effects of the transiently executed code are committed.

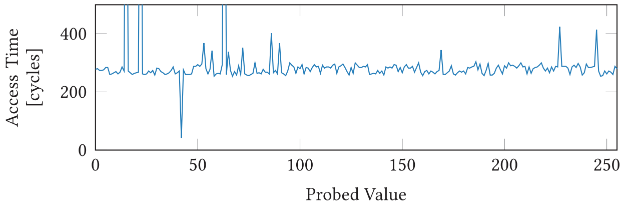


Figure 1: Access times to the probing array during the execution of Listing 1. The dip at 42 matches the value from the store buffer.

Recovering the Leaked Data. Using Flush+Reload, the attacker can recover the leaked value from the cache-based covert channel (Line 14). Figure 1 shows the results of measured access times to the look-up-table (`lut`) on a Meltdown-resistant i9-9900K CPU. As the figure illustrates, the typical access time to an array element is above 200 cycles, except for element 42, where the access time is well below 100 cycles. We note that this position matches the secret value written to `victim_page`. Hence, the code can recover the value without directly reading it.

Reading Writes From Other Contexts. Since there is no requirement for the upper address bits to match, the WTF shortcut allows any application to read the entire contents of the store buffer. Such behavior can be particularly dangerous if the store buffer contains data from other contexts. We discuss this in more detail in Section 4.

3.2 Data Bounce

Our second attack primitive, Data Bounce, exploits that storing to or forwarding from the store buffer lacks a write-permission check for the store address, e.g., for read-only memory and kernel memory. Under normal operating conditions, the full physical address is required for a valid store buffer entry. The store buffer entry is already reserved when the corresponding μOPs enter the reorder buffer. However, the store can only be correctly forwarded if there is a full virtual address or full physical addresses of the store's target are known [42]. This is no contradiction to the previously described observation, namely that stores can be incorrectly forwarded, e.g., in the case of partial address matches. Still, in Data Bounce we deliberately attempt to have a full virtual address match. We observe that virtual addresses without a valid mapping to physical addresses are not forwarded to subsequent loads to the same virtual address.

The basic idea of Data Bounce is to check whether a potentially illegal data write is forwarded to a data load from the same address. If the store-to-load forwarding is successful for a chosen address, we know that the chosen address can be resolved to a physical address. If done naively, such a test would destroy the value at addresses which the user can write to. Thus, we only test the store-to-load forwarding for an address in the transient-execution domain, i.e., the write is never committed architecturally.

```
① mov (0) → $dummy
② mov $x → (p)
③ mov (p) → $value
④ mov ($mem + $value * 4096) → $dummy
```

Figure 2: Data Bounce writes a known value to an accessible or inaccessible memory location, reads it back, encodes it into the cache, and finally recovers the value using a Flush+Reload attack. If the recovered value matches the known value, the address is backed by a physical page.

Figure 2 illustrates the basic principle of Data Bounce. First, we start transient execution by generating an exception and catching it (①). Alternatively, we can use any of the mechanisms mentioned in Section 3.1 to suppress the exception. For a chosen address p, we store a chosen value x using a simple data store operation (②). Subsequently, we read the value stored at address p (③) and encode it in the cache (④), as done for WTF (Section 3.1). We can now use Flush+Reload to recover the stored value, and distinguish two different cases as follows.

Store-to-Load Forwarding. If the value read from p is x, i.e., the x-th page of mem is cached, the store was forwarded to the load. Thus, we know that p is backed by a physical page. The choice of the value x is of no importance for Data Bounce. Even in the unlikely case that p already contains the value x, and the CPU reads the stale value from memory instead of the previously stored value x, we still know that p is backed by a physical page.

No Store-to-Load Forwarding. If no page of mem is cached, the store was not forwarded to the subsequent load. The cause of this could be either temporary or permanent. If a physical page does not back the virtual address, store-to-load forwarding always fails, i.e., even retrying the experiment will not be successful. Temporary causes for failure include interrupts, e.g., from the hardware timer, and errors in distinguishing cache hits from cache misses (e.g., due to power scaling). However, we find that if Data Bounce repeatedly fails for addr, the most likely cause is that addr is not backed by a physical page.

Breaking ASLR. In summary, if a value "bounces back" from a virtual address, the virtual address must be backed by a physical page. This effect can be exploited within the virtual address space of a process, e.g., to find which virtual addresses are mapped in a sandbox (cf. Section 6.2). On CPUs where Meltdown is mitigated in hardware, KAISER [25] is not enabled, and the kernel is again mapped in the virtual address space of processes [16]. In this case, we can also apply Data Bounce to kernel addresses. Even though we cannot access the data stored at the kernel address, we still can detect whether a physical page backs a particular kernel address. Thus, Data Bounce can still be used to break KASLR (cf. Section 6.1) on processors with in-silicon patches against Meltdown.

Handling Abnormal Addresses. We note that there are some cases where store forwarding happens without a valid mapping. However, these cases do not occur under normal operating conditions, hence we can ignore them for the purpose of Data Bounce. We discuss these conditions in Section 5.

3.3 Fetch+Bounce

Our third attack primitive, Fetch+Bounce, augments Data Bounce with an additional interaction between the TLB and the store buffer, allowing us to detect recent usage of virtual pages.

With Data Bounce it is easy to distinguish valid from invalid addresses. However, its success rate (i.e., how often Data Bounce has to be repeated) directly depends on which translations are stored in the TLB. Specifically, we observe cases where store-to-load forwarding fails when the mapping of the virtual address is not stored in the TLB. However, in other cases, when the mapping is already known, the store is successfully forwarded to a subsequent load. With Fetch+Bounce, we further exploit this TLB-related side-channel information by analyzing the success rate of Data Bounce.

```
① for retry = 0...2
     mov $x → (p)
②    mov (p) → $value
     mov ($mem + $value * 4096) → $dummy
③    if flush_reload($mem + $x * 4096) then break
```

Figure 3: Fetch+Bounce repeatedly executes Data Bounce. If Data Bounce succeeds on the first try, the address is in the TLB. If it succeeds on the second try, the address is valid but not in the TLB.

With Fetch+Bounce, we exploit that Data Bounce succeeds immediately if the mapping for the chosen address is already cached in the TLB. Figure 3 shows how Fetch+Bounce works. The basic idea is to repeat Data Bounce (②) multiple times (①). There are three possible scenarios, which are also illustrated in Figure 4.

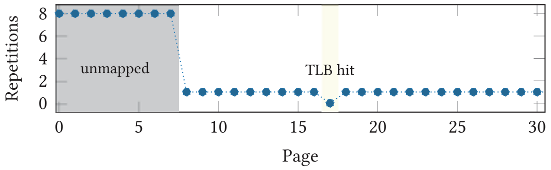


Figure 4: Mounting Fetch+Bounce on a virtual memory range allows to clearly distinguish mapped from unmapped addresses. Furthermore, for every page, it allows to distinguish whether the address translation is cached in the TLB.

TLB Hit. If the store's address is in the TLB, Data Bounce succeeds immediately, aborting the loop (③). Thus, retry is 0 after the loop.

TLB Miss. If the store's address is valid but is not in the TLB, Data Bounce fails in the first attempt, as the physical address needs to be resolved before store-to-load forwarding. As this creates a new TLB entry, Data Bounce succeeds in the second attempt (i.e., retry is 1). Note that this contradicts the official documentation saying that "transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs" [38].

Invalid Address. If the address cannot be fetched to the TLB, store-to-load forwarding fails and retry is larger than 1.

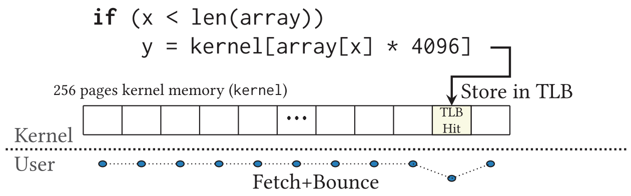
Just like Data Bounce, Fetch+Bounce can also be used on kernel addresses. Hence, with Fetch+Bounce we can deduce the TLB caching status for kernel virtual addresses. The only requirement is that the virtual address is mapped to the attacker's address space.

Fetch+Bounce is not limited to the data TLB (dTLB), but can also leak information from the instruction TLB (iTLB). Thus, in addition to recent data accesses, it is also possible to detect which (kernel) code pages have been executed recently.

One issue with Fetch+Bounce is that the test loads valid addresses to the TLB. For a real-world attack (cf. Section 7) this side effect is undesired, as measurements with Fetch+Bounce destroy the secret-dependent TLB state. Thus, to use Fetch+Bounce repeatedly on the same address, we must evict the TLB between measurements, e.g., using the strategy proposed by Gras et al. [21].

3.4 Speculative Fetch+Bounce

Our fourth attack primitive, Speculative Fetch+Bounce, augments Fetch+Bounce with transient-execution side effects on the TLB. The TLB is also updated during transient execution [70]. That is, we can even observe *transient* memory accesses with Fetch+Bounce.



```
if (x < len(array))
    y = kernel[array[x] * 4096]
```

Figure 5: Speculative Fetch+Bounce allows an attacker to use Spectre gadgets to leak data from the kernel, by encoding them in the TLB.

As a consequence, Speculative Fetch+Bounce poses a novel way to exploit Spectre. Instead of using the cache as a covert channel in a Spectre attack, we leverage the TLB to encode the leaked data. The advantage of Speculative Fetch+Bounce over the original Spectre attack is that there is no requirement for shared memory between user and kernel space. The attacker only needs control over an array index to leak arbitrary memory contents from the kernel. Figure 5 illustrates the encoding of the data, which is similar to the original Spectre attack [47]. Depending on the value of the byte to leak, we access one out of 256 pages. Then, Fetch+Bounce is used to detect which of the pages has a valid translation cached in the TLB. The cached TLB entry directly reveals the leaked byte.

4 BREAKING KERNEL ISOLATION

In this section, we show how to use the WTF shortcut to read data across security domains. We show leakage from the kernel to user space. Finally, Section 4.3 shows leakage from aborted TSX transactions.

4.1 Leaking Memory Writes from the Kernel

We start with a contrived scenario to evaluate an attacker's ability to recover kernel writes. Our proof-of-concept implementation consists of two components. The first is a kernel module that writes to a predetermined virtual address in a kernel page. The second is a user application that exploits the WTF shortcut using a faulty load that matches the page offset of the kernel store. The user application thus retrieves the data written by the kernel. We now describe these components.

The Kernel Module. Our kernel module performs a sequence of write operations, each to a different page offset in a different kernel page. These pages, like other kernel pages, are not directly accessible to user code. On older processors, such addresses may be accessible indirectly via Meltdown. However, we do not exploit this and assume that the user code does not or cannot exploit Meltdown.

The Attacker Application. The attacker application aims to retrieve kernel information that would normally be inaccessible from outside the kernel. The code first uses the `mprotect` system call to revoke access to an attacker-controlled page. Note that `mprotect` manipulates associated page table entry by clearing the present bit and applying PTE inversion [13], to cause the physical page frame number to be invalid.

The attacker application then invokes the kernel module to perform the kernel writes and afterward attempts to recover the values written by the kernel. To do this, the attacker performs a faulty load from his own protected page and transiently leaks the value through a covert cache channel.

Increasing the Window for the Faulty Load. Using WTF, we can read kernel writes even if the kernel only performed a *single* write before returning to the user. However, such an attack succeeds with low probability, and in most cases, the attack fails at reading the value stored by the kernel. We believe that the cause of the failure is that by the time the system switches from kernel to user mode, the store buffer is drained. Because store buffer entries are processed in order [3, 4, 33, 44], we can increase the time to drain the store buffer by performing a sequence of unrelated store operations in the attacker application or in the kernel module before the store whose value we would like to extract.

Experimental Evaluation. To evaluate the accuracy of our attack at recovering kernel writes, we design the following experiment. First, the kernel performs some number of single-byte store operations to different addresses. The kernel then performs an additional and last store to a target address, where we would like to recover the value written by this store. Finally, the kernel module returns to user space.

We evaluate the accuracy of our attack in Figure 6. The horizontal axis indicates the number of stores performed in the kernel module

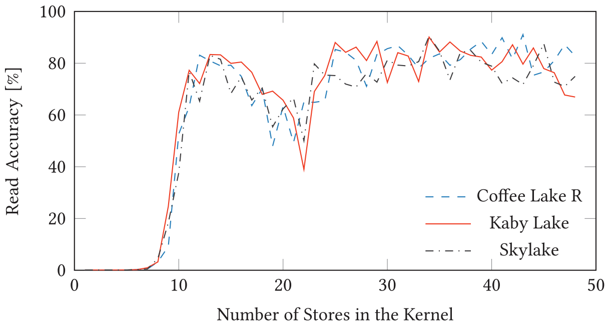


Figure 6: The success rate when recovering kernel values from user space as a function of the number of kernel stores.

(including the last targeted store), and the vertical axis is the success rate. For each data point, we tested the attack on all possible page offsets for the last kernel write, 100 times for each offset, reporting the success rate.

For our evaluation, we use three Intel machines with Skylake (i7-6700), Kaby Lake (i7-7600) and Coffee Lake R (i9-9900K) processors, each running a fully updated Ubuntu 16.04. As Figure 6 shows, the kernel module needs to perform 10 or more writes (to different addresses) before returning to the user for the attack to succeed at recovering the last kernel store with 50–80% success rate. Finally, recovering values from a kernel performing a single write before returning can be done with a success rate of 0.05%.

On processors vulnerable to Meltdown, disabling the KAISER patch exposes the machine to Meltdown attacks on the kernel. However, on the Coffee Lake R processor, which includes hardware countermeasures for Meltdown, KAISER is disabled by default. In particular, the experiments for this processor in Figure 6 are with the default Ubuntu configuration. This means that the presence of the hardware countermeasures in Intel's latest CPU generations led to software behavior that is more vulnerable to our attack compared to systems with older CPUs.

4.2 Attacking the AES-NI Key Schedule

We now proceed to a more realistic scenario. Specifically, we show how the WTF shortcut can leak to a user the secret encryption keys processed by the kernel.

The Linux kernel cryptography API supports several standard cryptographic schemes that are available to third-party kernel modules and device drivers which need cryptography. For example, the Linux key management facility and disk encryption services, such as eCryptfs [32], heavily rely on this cryptographic library.

To show leakage from the standard cryptographic API, we implemented a kernel module that uses the library to provide user applications with an encryption oracle. We further implemented a user application that uses the kernel module. The AES keys that the kernel module uses are only stored in the kernel and are never shared with the user. However, our application exploits the WTF shortcut to leak these keys from the kernel. We now describe the attack in further details.

AES and AES-NI. A 128-bit AES encryption or decryption operation consists of 10 *rounds*. The AES key schedule algorithm expands the AES master key to generate a separate 128-bit *subkey* for each of these rounds. An important property of the key scheduling algorithm is that it is reversible. Thus, given a subkey, we can reverse the key scheduling algorithm to recover the master key. For further information on AES, we refer to FIPS [59].

Since encryption is a performance-critical operation and to protect against side-channel attacks [62], recent Intel processors implement the AES-NI instruction set [31], which provides instructions that perform parts of the AES operations. In particular, the AESKEYGENASSIST instruction performs part of the key schedule algorithm.

Key Scheduling in Linux. The Linux implementation stores the master key and the 10 subkeys in consecutive memory locations. With each subkey occupying 16 bytes, the total size of the expanded key is 176 bytes. Where available, the Linux kernel cryptography API uses AES-NI for implementing the AES functionality. Part of the code that performs key scheduling for 128-bit AES appears in Listing 2. Lines 1 and 3 invoke AESKEYGENASSIST to perform a step of generating a subkey for a round. The code then calls the function _key_expansion_128, which completes the generation of the subkey. The process repeats ten times, once for each round. (To save space we only show two rounds.)

```
 1 aeskeygenassist $0x1, %xmm0, %xmm1
 2 callq  <_key_expansion_128>
 3 aeskeygenassist $0x2, %xmm0, %xmm1
 4 callq  <_key_expansion_128>
 5 ...
 6 <_key_expansion_128>:
 7 pshufd $0xff,%xmm1,%xmm1
 8 shufps $0x10,%xmm0,%xmm4
 9 pxor   %xmm4,%xmm0
10 shufps $0x8c,%xmm0,%xmm4
11 pxor   %xmm4,%xmm0
12 pxor   %xmm1,%xmm0
13 movaps %xmm0,(%r10)
14 add    $0x10,%r10
15 retq
```

Listing 2: AES-NI key schedule.

_key_expansion_128 starts at Line 6. It performs the operations needed to complete the generation of a 128-bit AES subkey. It then writes the subkey to memory (Line 13) before advancing the pointer to prepare for storing the next subkey (Line 14) and returning.

Finding the Page Offset. We aim to capture the key by leaking the values stored in Line 13. For that, the user application repeatedly invokes the kernel interface that performs the key expansion as part of setting up an AES context. Because the AES context is allocated dynamically, its address depends on the state of the kernel's memory allocator at the time the context is allocated. This prevents immediate use of our attack because the attacker does not know where the subkeys are stored.

We use the WTF shortcut to recover the page offset of the AES context. Specifically, the user application scans page offsets. For each offset, it asks the kernel module to initialize the AES context. It then performs a faulty load from a protected page at the scanned offset and checks if any data leaked. To reduce the number of scanned offsets, we observe that, as described above, the size of the expanded key is 176 bytes. Hence, we can scan at offsets that are

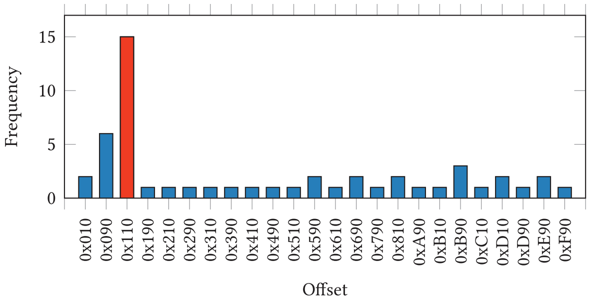


Figure 7: Frequency of observed leaked values. We note that offset 0x110 shows more leakage than others. Confirming against the ground truth, we find that all the leaked values at that offset match the subkey byte.

EXHIBIT 3
Page 7 of 16

128 bytes apart and have the confidence that at least one of these offsets falls within the expanded key. Indeed, running the attack for five minutes, we get Figure 7. The figure shows the number of leaked values at each offset over the full five minutes. We note the spike at offset 0x110. We compare the result to the ground truth and find that the expanded key indeed falls at offset 0x110. We further find that the leaked byte matches the value at page offset 0x110.

Key Recovery. Once we find one offset within the expanded key, we know that neighboring offsets also fall within the expanded key, and we can use the WTF shortcut to recover the other key bytes. We experiment with 10 different randomly selected keys and find that we can recover the 32 bytes of the subkeys of the two final rounds (rounds 9 and 10) without errors within two minutes. Reversing the key schedule on the recovered data gives us the master key.

4.3 Reading Data from TSX Transactions

Intel TSX guarantees that computation inside a transaction is either fully completed, having its outputs committed to memory or fully reverted if the transaction fails for any reason. In either case, TSX guarantees that intermediate computation values (which are not part of the final output) never appear in process memory. Building on this property, Guan et al. [30] suggest using TSX to protect cryptographic keys against memory disclosure attacks by keeping the keys encrypted, decrypting them inside a transaction, and finally zeroing them out before finishing the transaction. This way, Guan et al. [30] ensure that the decrypted keys never appear in the process' main memory, making them safe from disclosure.

Exploiting the WTF shortcut and Data Bounce against TSX transactions, we are able to successfully recover intermediate values, and hidden control flow from within completed or aborted TSX transactions.

5 INVESTIGATING STORE BUFFER LEAKAGE

In this section, we form a foundation for understanding the underlying mechanisms involved in WTF and Data Bounce. We start with a discussion of microcode assists, a hitherto uninvestigated cause for transient execution that extends the Meltdown vs. Spectre classification of Canella et al. [10]. We continue with the investigation of the underlying conditions for both WTF and Data Bounce. We conclude by testing our attacks in multiple processor generations.

5.1 Microcode Assists

μOPs are typically implemented in hardware. However, when complex processing is required for rare corner cases, a hardware implementation may not be cost-effective. Instead, if such a case occurs during the execution of a μOP, the μOP is *re-dispatched*, i.e., sent back to the dispatch queue for execution, together with a *microcode assist*, a microcode procedure that handles the more complex scenario. Cases in which microcode assists can occur include handling of subnormal floating point numbers, the use of REP MOV instruction to copy large arrays, and others [14, 42].

```
1  char* victim_page = mmap(..., PAGE_SIZE, ...);
2  char* attacker_page = mmap(..., PAGE_SIZE, ...);
3
4  offset = 7;
5  victim_page[offset] = 42;
6
7  clear_access_bit(attacker_page);
8  memory_access(lut + 4096 * attacker_page[offset]);
9
10 for (i = 0; i < 256; i++) {
11   if (flush_reload(lut + i * 4096)) {
12     report(i);
13   }
14 }
```

Listing 3: Exploiting the WTF Shortcut with microcode assists. Note that no fault suppressison is required.

Microcode-Assisted Memory Accesses. According to an Intel patent [20], when the processor handles a memory access (load or store) it needs to translate the virtual address specified by the program to the corresponding physical address. For that, the processor first consults the Data Translation Look-aside Buffer (dTLB), which caches the results of recent translations. In the case of a page miss, i.e., when the virtual address is not found in the dTLB, the *page miss handler* (PMH) attempts to consult the page table to find the translation. In most cases, this translation can be done while the μOP is speculative. However, in some cases, the page walk has side effects that cannot take place until the μOP retires. Specifically, store operations should mark pages as dirty and all memory operations should mark pages as accessed. Performing these side effects while the μOP is speculative risks generating an architecturally-visible side effect for a transient μOP. (Recall that the processor cannot determine whether speculative μOPs will retire or not.) At the same time, recording all the information required for setting the bits on retirement would require a large amount of hardware that will only be used in relatively rare cases. Thus, to handle these cases, the processor re-dispatches the μOP and arranges for a microcode assist to set the bits when the μOP retires. See the patent [20] for further details on the process.

Recall (Section 2.2) that Canella et al. [10] classify transient-execution attacks based on the cause of transient execution. Spectre-type attacks are caused by misprediction of data or control flow, whereas Meltdown-type attack are caused by transient execution beyond a fault. As described above, a μOP re-dispatch occurring as part of handling microcode assists also causes transient execution.

Assist-based WTF. To test the effects of microcode assists on the WTF shortcut, we use the code in Listing 3. To mark `attack_page` as not accessed (Line 7), we can either use the Linux idle page tracking interface [17] or the page table manipulation options in SGX-Step [75]. Using these methods for clearing the accessed bit requires elevated privileges. However, some operating systems may clear the accessed bit regularly or upon paging pressure, obviating the need for root access. Furthermore, because microcode assists do not generate faults, we do not need fault suppression, and remove the TSX transaction.

Assist-based vs. Meltdown-type. Canella et al. [10] list several properties of Meltdown-type attacks. Assist-based transient execution shares *some* properties with Meltdown-type techniques. Specifically, it relies on deferred termination of a μOP to bypass hardware security barriers and attacks based on it can be mitigated by preventing the original leak. However, unlike Meltdown-type techniques, assists do not rely on architectural exceptions. Consequently, no fault suppression techniques are required. Thus, assist-based techniques represent a new cause to trigger transient execution. In a concurrent work, Schwarz et al. [68] also identify that assists result in transient execution. They extend the definition of Meltdown-type to include microcode assists, which they describe as "(microarchitectural) faults".

5.2 Analyzing WTF

In this section we deepen our investigation of WTF by considering various causes for faulting loads and the fault suppression used. Particularly, for fault-suppression we experiment with both TSX-based suppression and with using branch misprediction. We ran our experiments on three Intel processors: Coffee Lake R i9-9900K, Kaby Lake i7-7600U, and Skylake i7-6700. The only exception is Protection Keys, which are not available on these processors, and were tested on a Xeon Silver 4110 processor. To the best of our knowledge, no Coffee Lake R processor supports Protection Keys. We summarize the results in Table 1.

We use the toy example in Listing 1 with multiple combinations of causes of illegal loads and fault-suppression mechanisms for the load. Following the analysis by Canella et al. [10], we systematically investigated the following exception types as causes for illegal loads.

Non-Canonical. We found that the easiest way to trigger WTF is by provoking a general protection exception (#GP) when accessing a non-canonical address outside of the valid range of virtual addresses represented by the processor [38]. Our experiments show that this technique works reliably on all tested processors and exception suppression mechanisms.

Supervisor Fault. We note that on Linux even when KPTI is enabled, some kernel code pages remain mapped in a user process's address space (see Section 6.1) and can hence be used to provoke faulting loads on kernel addresses (raising a #PF exception). We found that supervisor page faults can be successfully abused to trigger WTF on all tested processors and exception suppression mechanisms.

Supervisor Mode Access Prevention (SMAP). For completeness, we also tested whether WTF can be triggered by SMAP features [38]. For this experiment, we explicitly dereference a user space pointer in kernel mode such that SMAP raises a #PF exception. We observed that SMAP violations may successfully trigger the WTF shortcut on all tested processors and exception suppression mechanisms. While we do not consider this to be an exploitable attack scenario, SMAP was to the best of our knowledge previously considered to be immune to any Meltdown-type effects [10].

Protection Key Fault. We investigated triggering WTF via reading from pages marked as unreadable using Intel's Protection Key mechanism [38], which also raises a page fault (#PF) exception. We found that Protection Key violations may successfully trigger WTF on the tested Xeon processor with all exception suppression mechanisms.

Misalignment in Advanced Vector Extensions (AVX). We investigated whether WTF may also be triggered by general protection fault exceptions (#GP) generated by misaligned AVX load instructions [38]. Interestingly, we found that this technique works exclusively using TSX exception suppression on recent Coffee Lake R processors.

Non-Present Fault and Coffee Lake R Regression. We investigated triggering WTF from non-present pages both with and without PTE inversion [13]. In our experiments, we created the former using the `mprotect` system call with the permission set to `PROT_NONE`, and the latter by unmapping the target page using the `munmap` system call. While dereferencing non-present pages always causes the CPU to raise a page fault (#PF) exception, we noticed a troubling *regression* in Intel's newest Coffee Lake R architecture. Where, unlike earlier generations, we can successfully trigger the WTF shortcut on Coffee Lake R processors when accessing a page marked as non-present from within a TSX transaction.

Fault Suppression	**TSX**		**Misprediction**	
Architecture	Pre CL R	CL R	Pre CL R	CL R
Non-canonical	✓	✓	✓	✓
Kernel pages	✓	✓	✓	✓
User pages with SMAP	✓	✓	✓	✓
Protection keys	✓	N/A	✓	N/A
AVX misalignment	✗	✓⃝	✗	✗
Not present with PTE inversion	✗	✓⃝	✗	✗
Not present without PTE inversion	✗	✗	✗	✗

Table 1: Evaluating the WTF shortcut using different fault-inducing and fault-suppression mechanisms on Intel architectures before Coffee Lake R (pre CL R) and on Coffee Lake R (CL R). ✓ and ✗ indicate attack success. ✓⃝ and ✗⃝ indicate behavior change in Coffee Lake R.

Interestingly, our investigation revealed that the behavior in the case of non-present pages depends on the contents of the page-frame number in the page-table entry. Specifically, we have only seen WTF working on Coffee Lake R when the page-frame number in the PTE refers to an invalid page frame or to EPC pages. We note that widely deployed PTE inversion [13] software mitigations for Foreshadow modify the contents of the page-frame number for pages protected with `mprotect` to point to invalid page frames (i.e., not backed by physical DRAM). Our experiments show that the WTF shortcut is only triggered when loading from these pages from within a TSX transaction, whereas WTF seems not to be activated when dereferencing unmapped pages with valid page-frame numbers, both inside or outside TSX. We suspect that the CPU inhibits some forms of transient execution within branch mispredictions while allowing them in TSX transactions.

5.3 Analyzing Store-to-Leak

Store-to-Leak exploits address resolution logic in the store buffer. Namely, that in case of a full virtual address match between a load and a prior store, store-to-load forwarding logic requires that the load operation may only be unblocked *after* the physical address of the prior store has been resolved [33]. In this case, if the tested virtual address has a valid mapping to a physical address, whether accessible to the user or not, the store is forwarded to the load.

Recovering Information About Address Mapping. The success of Store-to-Leak, therefore, provides two types of side-channel information on the address mapping of the tested virtual address. First, we observed that Data Bounce reliably triggers forwarding in the first attempt when writing to addresses that have a valid virtual mapping in the TLB. Secondly, when writing to addresses

CPU	Data Bounce	Fetch+Bounce	Speculative Fetch+Bounce	WTF
Pentium 4 531	✓	✗	✗	✗
i5-3230M	✓	✓	✓	✓
i7-4790	✓	✓	✓	✓
i7-6600U	✓	✓	✓	✓
i7-6700K	✓	✓	✓	✓
i7-8650U	✓	✓	✓	✓
i9-9900K	✓	✓	✓	✓
E5-1630 v4	✓	✓	✓	✓

Table 2: Attack techniques and processors we evaluated.

that have a valid physical mapping but are currently not cached in the TLB, we found that Store-to-Leak still works after multiple repeated Data Bounce attempts. Overall, as Data Bounce never performs forwarding for unmapped addresses that do not have a valid physical mapping, the attacker may learn whether an address has a valid physical mapping and whether this mapping was cached inside the TLB.

Finally, we also observed two exceptions to the above, in which Store-to-Leak may still trigger forwarding for addresses that are not backed by a valid virtual address mapping. We now proceed to explain these exceptions and how they affect Store-to-Leak.

The Strange Case of Non-Canonical Addresses. First, we experimentally confirmed that on all tested processors, Data Bounce forwards data when writing to and subsequently reading from a non-canonical address. This behavior is peculiar since dereferencing non-canonical addresses always generates a general protection fault (#GP) as these addresses are invalid by definition and can never be backed by a physical address [38]. We note, however, that all attack techniques based on Store-to-Leak only use canonical addresses and our attacks are hence not hindered by these observations.

Non-Present Pages and Coffee Lake R. Secondly, we noticed a different behavior in Intel's newest Coffee Lake R architecture. Where, unlike earlier generations, we can successfully trigger Data Bounce when accessing a non-present page from within a TSX transaction. Notably, we have only seen Store-to-Leak forwarding for non-present pages on Coffee Lake R when the page-frame number in the PTE refers to an invalid page frame, and Data Bounce executes within a TSX transaction. We have not seen this behavior with any other fault-suppression primitive or on any other TSX-enabled CPU. Furthermore, note that we never encountered an inverted kernel page table entry, but instead observed that unmapped kernel pages always have an all-zero page-frame number. Hence, the Store-to-Leak attacks described in this paper are not affected by these observations.

5.4 Environments

We evaluated all attack techniques on multiple Intel CPUs. All attack primitives worked on all tested CPUs, which range from the Ivy Bridge architecture (released 2012) to Whiskey Lake and Coffee Leak R (both released end of 2018). The only exception is a Pentium 4 Prescott CPUs (released 2004), on which only Data Bounce works. Table 2 contains the list of CPUs we used for evaluation.

Next, the attack primitives are not limited to the Intel's Core architecture but also work on Intel's Xeon architecture. Thus, our attacks are not limited to consumer devices, but can also be used in the cloud. Furthermore, our attacks even work on CPUs with silicon fixes for Meltdown and Foreshadow, such as the i7-8565U and i9-9900K [16]. Finally, we were unable to reproduce our attack primitives on AMD and ARM CPUs, limiting the attacks to Intel.

6 ATTACKS ON ASLR

In this section, we evaluate our attack on ASLR in different scenarios. As Data Bounce can reliably detect whether a physical page backs a virtual address, it is well suited for breaking all kinds of ASLR. In Section 6.1, we show that Data Bounce is the fastest way and most reliable side-channel attack to break KASLR on Linux, and Windows, both in native environments as well as in virtual machines. In Section 6.2, we describe that Data Bounce can even be mounted from JavaScript to break ASLR of the browser.

6.1 Breaking KASLR

We now show that Data Bounce can reliably break KASLR. We evaluate the performance of Data Bounce in two different KASLR breaking attacks, namely de-randomizing the kernel base address as well as finding and classify modules based on detected size.

De-randomizing the Kernel Base Address. On Linux systems, KASLR had been supported since kernel version 3.14 and enabled by default since around 2015. As Jang et al. [45] note, the amount of entropy depends on the kernel address range as well as on the alignment size, which is usually a multiple of the page size.

We verified this by checking `/proc/kallsyms` across multiple reboots. With a kernel base address range of 1 GB and a 2 MB alignment, we get 9 bits of entropy, allowing the kernel to be placed at one of 512 possible offsets.

Using Data Bounce, we now start at the lower end of the address range and test all of the 512 possible offsets. If the kernel is mapped at a tested location, we will observe a store-to-load forwarding identifying the tested location as having a valid mapping to a physical address. Table 3 shows the performance of Data Bounce in de-randomizing kernel ASLR. We evaluated our attack on both an Intel Skylake i7-6600U (without KAISER) and a new Intel Coffee Lake i9-9900K that already includes fixes for Meltdown [51] and Foreshadow [74]. We evaluated our attack on both Windows and Linux, achieving similar results.

For the evaluation, we tested 10 different randomizations (i.e., 10 reboots). In each, we try to break KASLR 100 times, giving us a total of 1000 samples. For evaluating the effectiveness of our attack, we use the F1-score. On the i7-6600U and the i9-9900K, the F1-score for finding the kernel ASLR offset is 1 when testing every offset a single time, indicating that we always find the correct offset. In terms of performance, we outperform the previous state of the art [45] even though our search space is 8 times larger. Furthermore, to evaluate the performance on a larger scale, we tested a single offset 100 million times. In that test, the F1-score was 0.9996, showing that Data Bounce virtually always works. The few misses that we observe are possibly due to the store buffer being drained or that our test program was interrupted.

Finding and Classifying Kernel Modules. The kernel reserves 1 GB for modules and loads them at 4 kB-aligned offset. In a first step, we can use Data Bounce to detect the location of modules by iterating over the search space in 4 kB steps. As kernel code is

Processor	Target	#Retries	#Offsets	Time	F1-Score
Skylake (i7-6600U)	base	1	512	72 µs	1
	direct-physical	3	64000	13.648 ms	1
	module	32	262144	1.713 s	0.98
Coffee Lake (i9-9900K)	base	1	512	42 µs	1
	direct-physical	3	64000	8.61 ms	1
	module	32	262144	1.33 s	0.96

Table 3: Evaluation of Data Bounce in finding the kernel base address, its direct-physical map, and the kernel modules. Number of retries refers to the maximum number of times an offset is tested, and number of offsets denotes the maximum number of offsets that need to be tried.

always present and modules are separated by unmapped addresses, we can detect where a module starts and ends. In a second step, we use this information to estimate the size of all loaded kernel modules. The world-readable `/proc/modules` file contains information on modules, including name, size, number of loaded instances, dependencies on other modules, and load state. For privileged users, it additionally provides the address of the module. We correlate the size from `/proc/modules` with the data from our Data Bounce attack and can identify all modules with a unique size. On the i7-6600U, running Ubuntu 18.04 (kernel version 4.15.0-47), we have a total of 26 modules with a unique size. On the i9-9900K, running Ubuntu 18.10 (kernel version 4.18.0-17), we have a total of 12 modules with a unique size. Table 3 shows the accuracy and performance of Data Bounce for finding and classifying those modules.

Breaking KASLR with the KAISER Patch. As a countermeasure to Meltdown [51], OSs running on Intel processors prior to Coffee Lake R have deployed the KAISER countermeasure, which removes the kernel from the address space of user processes (see Figure 8 (bottom)). To allow the process to switch to the kernel address space, the system leaves at least one kernel page in the address space of the user process. Because the pages required for the switch do not contain any secret information, there is no need to hide them from Meltdown [12].

However, we observed that the pages that remain in the user space are randomized using the same offset as KPTI. Hence, we can use Data Bounce to de-randomize the kernel base address even with KPTI enabled. To the best of our knowledge, we are the first to demonstrate KASLR break with KPTI enabled. Finally, we note that on CPUs with hardware Meltdown mitigation, our KASLR break is more devastating, because we can de-randomize not only the kernel base address but also the kernel modules

6.2 Recovering Address Space Information from JavaScript

In addition to unprivileged native applications, Data Bounce can also be used in JavaScript to leak partial information on allocated and unallocated addresses in the browser. This information can potentially lead to breaking ASLR. In this section, we evaluate the performance of Data Bounce from JavaScript running in a modern browser. We conducted this evaluation on Google Chrome 70.0.3538.67 (64-bit) and Mozilla Firefox 66.0.2 (64-bit).

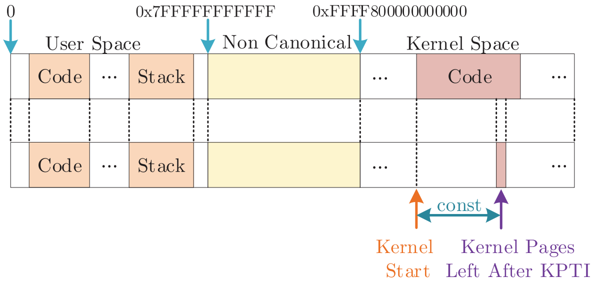


Figure 8: (Top) Address space with KASLR but without KAISER. (Bottom) User space with KASLR and KAISER. Most of the kernel is not mapped in the process's address space anymore.

There are two main challenges for mounting Data Bounce from JavaScript. First, there is no high-resolution timer available. Therefore, we need to build our own timing primitive. Second, as there is no flush instruction in JavaScript, Flush+Reload is not possible. Thus, we have to resort to a different covert channel for bringing the microarchitectural state to the architectural state.

Timing Primitive. To measure timing with a high resolution, we rely on the well-known use of a counting thread in combination with shared memory [22, 69]. As Google Chrome has re-enabled `SharedArrayBuffers` in version 67 [2], we can use the existing implementations of such a counting thread. In Firefox, we emulated this behavior by manually enabling `SharedArrayBuffers`.

In Google Chrome, we can also use the `BigUint64Array` to ensure that the counting thread does not overflow. This improves the measurements compared to the `Uint32Array` used in previous work [22, 69] as the timestamp is increasing strictly monotonically. In our experiments, we achieve a resolution of 50 ns in Google Chrome, which is sufficient to distinguish a cache hit from a miss.

Covert Channel. As JavaScript does not provide a method to flush an address from the cache, we have to resort to eviction, as shown in previous work [22, 47, 61, 69, 77]. Thus, our covert channel from the microarchitectural to the architectural domain, i.e., the decoding of the leaked value which is encoded into the cache, uses Evict+Reload instead of Flush+Reload.

For the sake of simplicity, we can also access an array 2–3 times larger than the last-level cache to ensure that data is evicted from the cache. For our proof-of-concept, we use this simple approach as it is robust and works for the attack. While the performance increases significantly when using targeted eviction, we would require 256 eviction sets. We avoid generating these eviction sets because the process is time-consuming and prone to errors.

Illegal Access. In JavaScript, we cannot access an inaccessible address architecturally. However, as all modern browsers use just-in-time compilation to convert JavaScript to native code, we can leverage speculative execution to prevent the fault. Hence, we rely on the same code as Kocher et al. [47] to speculatively access an out-of-bounds index of an array. This allows to iterate over the memory (relative from our array) and detect which pages are mapped and which pages are not mapped.

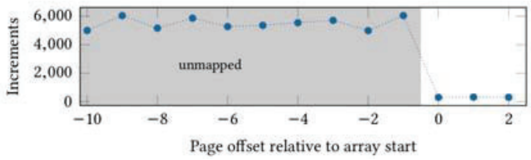


Figure 9: Data Bounce with Evict+Reload in JavaScript clearly shows whether an address (relative to a base address) is backed by a physical page and thus valid.

Full Exploit. When putting everything together, we can distinguish for every location relative to the start array, whether a physical page backs it or not. Due to the limitations of the JavaScript sandbox, especially due to the slow cache eviction, the speed is orders of magnitude slower than the native implementation, as it can be seen in Figure 9. Still, we can detect whether a virtual address is backed by a physical page within 450 ms, making Data Bounce also realistic from JavaScript.

7 FETCH+BOUNCE

Fetch+Bounce uses Data Bounce to spy on the TLB state and enables more powerful attacks as we show in this section. So far, most microarchitectural side-channel attacks on the kernel require at least some knowledge of physical addresses [65, 67]. Since physical addresses are not provided to unprivileged applications, these attacks either require additional side channels [26, 67] or have to blindly attack targets until the correct target is found [71].

With Fetch+Bounce we directly retrieve side-channel information for any target virtual address, regardless of the access permissions in the current privilege level. We can detect whether a virtual address has a valid translation in either the iTLB or dTLB, thereby allowing an attacker to infer whether an address was recently used.

Fetch+Bounce allows an attacker to detect recently accessed *data pages* in the current hyperthread. Moreover, an attacker can also detect *code pages* recently used for instruction execution in the current hyperthread. Next, as the measurement with Fetch+Bounce results in a valid mapping of the target address, we also require a method to evict the TLB. While this can be as simple as accessing (dTLB) or executing (iTLB) data on more pages than there are TLB entries, this is not an optimal strategy. Instead, we rely on the reverse-engineered eviction strategies from Gras et al. [21].

We first build an eviction set for the target address(es) and then loop Fetch+Bounce on the target address(es) to detect potential activity, before evicting the target address(es) again from iTLB and dTLB. Below, we demonstrate this attack on the Linux kernel.

7.1 Inferring Control Flow of the Kernel

The kernel is a valuable target for attackers, as it processes all inputs coming from I/O devices. Microarchitectural attacks targeting user input directly in the kernel usually rely on Prime+Probe [58, 61, 66, 67] and thus require recovery of physical address information.

With Fetch+Bounce, we do not require knowledge of physical addresses to spy on the kernel. In the following, we show that Fetch+Bounce can spy on any type of kernel activity. We illustrate this with the examples of mouse input and Bluetooth events.

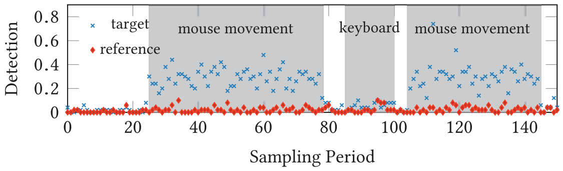


Figure 10: Mouse movement detection. The mouse movements are clearly detected. The USB keyboard activity does not cause more TLB hits than observed as a baseline.

As a proof of concept, we monitor the first 8 pages of a target kernel module. To obtain a baseline for the general kernel activity, and thus the TLB activity for kernel pages, we also monitor one reference page from a rarely-used kernel module (in our case `i2c_i801`). By comparing the activity on the 8 pages of the kernel module to the baseline, we determine whether the module is currently used or not. For best results, we use Fetch+Bounce with both the iTLB and dTLB. This makes the attack independent of the activity type in the module, i.e., there is no difference between data access and code execution. Our spy changes its hyperthread after each Fetch+Bounce measurement. While this reduces the attack's resolution, it allows to detect activity on all hyperthreads. Next, we sum the resulting TLB hits over a *sampling period* which consists of 5000 measurements, and then apply a basic detection filter to this sum by calculating the ratio between hits on the target and reference pages. If the number of hits on the target pages is above a sanity lower bound and above the number of cache hits on the reference page, i.e., above the baseline, then the page was recently used.

Detecting User Input. We now investigate how well Fetch+Bounce works for spying on input-handling code in the kernel. While [67] attacked the kernel code for PS/2 keyboards, we target the kernel module for USB human-interface devices, allowing us to monitor activity on a large variety of modern USB input devices.

We first locate the kernel module using Data Bounce as described in Section 6.1. With 12 pages (kernel 4.15.0), the module does not have a unique size among all modules but is 1 of only 3. Thus, we can either try to identify the correct module or monitor all of them.

Figure 10 shows the results of using Fetch+Bounce on a page of the `usbhid` kernel module. It can be clearly seen that mouse movement results in a higher number of TLB hits. USB keyboard input, however, seems to fall below the detection threshold with our simple method. Given this attack's low temporal resolution, repeated accesses to a page are necessary for clear detection. Previous work has shown that such an event trace can be used to infer user input, e.g., URLs [49, 61].

Detecting Bluetooth Events. Bluetooth events can give valuable information about the user's presence at the computer, e.g., connecting (or disconnecting) a device usually requires some form of user interaction. Tools, such as Windows' Dynamic Lock [57], use the connect and disconnect events to unlock and lock a computer automatically. Thus, these events are a useful indicator for detecting whether the user is currently using the computer, as well as serve as a trigger signal for UI redressing attacks.

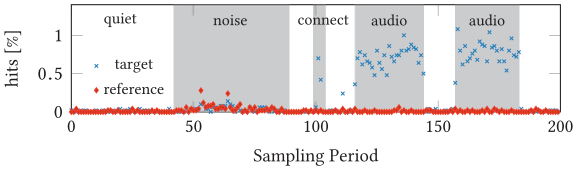


Figure 11: Detecting Bluetooth events by monitoring TLB hits via Fetch+Bounce on pages at the start of the `bluetooth` kernel module.

To spy on these events, we first locate the Bluetooth kernel module using Data Bounce. As the Bluetooth module is rather large (134 pages on kernel 4.15.0) and has a unique size, it is easy to distinguish it from other kernel modules.

Figure 11 shows a Fetch+Bounce trace while generating Bluetooth events. While there is a constant noise floor due to TLB collisions, we can see a clear increase in TLB hits on the target address for every Bluetooth event. After applying our detection filter, we can detect events such as connecting and playing audio over the Bluetooth connection with a high accuracy.

Our results indicate that the precision of the detection and distinction of events with Fetch+Bounce can be significantly improved. Future work should investigate profiling code pages of kernel modules, similar to previous template attacks [29].

8 LEAKING KERNEL MEMORY

In this section, we present Speculative Fetch+Bounce, a novel covert channel to leak memory using Spectre. Most Spectre attacks, including the original Spectre attack, use the cache as a covert channel to encode values leaked from the kernel [11, 35, 46–48, 54, 60, 70]. Other covert channels for Spectre attacks, such as port contention [9] or AVX [70] have since been presented. However, it is unclear how commonly such gadgets can be found and can be exploited in real-world software.

With Speculative Fetch+Bounce, we show how TLB effects on the store buffer (cf. Section 7) can be combined with speculative execution to leak kernel data. We show that any cache-based Spectre gadget can be used for Speculative Fetch+Bounce. As secret-dependent page accesses also populates the TLB, such a gadget also encodes the information in the TLB. With Data Bounce, we can then reconstruct which of the pages was accessed and thus infer the secret. While at first, the improvements over the original Spectre attack might not be obvious, there are two advantages.

Advantage 1: It requires less control over the Spectre gadget. First, for Speculative Fetch+Bounce, an attacker requires less control over the Spectre gadget. In the original Spectre Variant 1 attack, a gadget like `if ( index < bounds ) { y = oracle[ data[index] * 4096 ]; }` is required. There, an attacker requires full control over `index`, and also certain control over `oracle`. Specifically, the base address of `oracle` has to point to user-accessible memory which is shared between attacker and victim. Furthermore, the base address has to either be known or be controlled by the attacker. This limitation potentially reduces the number of exploitable gadgets.

Advantage 2: It requires no shared memory. Second, with Speculative Fetch+Bounce, we get rid of the shared-memory requirement. Especially on modern operating systems, shared memory is a limitation, as these operating systems provide stronger kernel isolation [25]. On such systems, only a few pages are mapped both in user and kernel space, and they are typically inaccessible from the user space. Moreover, the kernel can typically not access user space memory due to supervisor mode access prevention (SMAP). Hence, realistic Spectre attacks have to resort to Prime+Probe [73]. However, Prime+Probe requires knowledge of physical addresses, which is not exposed on modern operating systems.

With Speculative Fetch+Bounce, it is not necessary to have a memory region which is user accessible and shared between user and kernel. For Speculative Fetch+Bounce, it is sufficient that the base address of `oracle` points to a kernel address which is also mapped in user space. Even in the case of KPTI [53], there are still kernel pages mapped in the user space. On kernel 4.15.0, we identified 65536 such kernel pages when KPTI is enabled, and multiple gigabytes when KPTI is disabled. Hence, `oracle` only has to point to any such range of mapped pages. Thus, we expect that there are simpler Spectre gadgets which are sufficient to mount this attack.

Leaking Data. To evaluate Speculative Fetch+Bounce, we use a custom `ioctl` in the Linux kernel containing a Spectre gadget as described before. We were able to show that our proof-of-concept Spectre attack works between user and kernel in modern Linux systems, without the use of shared memory.

9 DISCUSSION AND COUNTERMEASURES

Intel recently announced [37] that new post-Coffee Lake R processors are shipped with silicon-level mitigations against WTF (MSBDS in Intel terminology). However, to the best of our knowledge, Intel did not release an official statement regarding Store-to-Leak mitigations. In this section, we discuss the widely deployed software and microcode mitigations released by Intel to address microarchitectural data sampling attacks [41]. We furthermore discuss the limitations of our analysis.

Leaking Stale Store Buffer Data. In this paper and our original vulnerability disclosure report, we focused exclusively on leaking *outstanding* store buffer entries in the limited time window after the kernel transfers execution to user space. That is, we showed that the WTF shortcut can be abused by unprivileged adversaries to leak in-flight data from prior kernel store instructions that have successfully retired but whose data has not yet been written out to the memory hierarchy. Hence, for our attacks to work, the stores must still be outstanding in the core's store buffer, and we are only able to recover at most the k most recent stores, where k is the store buffer size (cf. Appendix A for measurement of the store buffer size).

Concurrent to our work, Intel's analysis [41] of store buffer leakage revealed that WTF may furthermore be abused to leak *stale* data from older stores, even after the store data has been committed to memory, and the corresponding store buffer entry has been freed. This observation has profound consequences for defenses, as merely draining outstanding stores by serializing the instruction stream (e.g., using `mfence`) does *not* suffice to fully mitigate store buffer leakage.

Leaking Stores across HyperThreads. In Appendix A, we measured the size of the store buffer. We discover that when both logical CPUs on the same physical core are active, the store buffer is statically partitioned between the threads. Otherwise, a thread can use the entire store buffer. Consequently, one hardware thread will not be able to read writes performed by another thread running in parallel. However, Intel's analysis [41] describes that leakage may still occur when hardware threads go to sleep since stale store buffer entries from the other thread are reused, or when hardware threads wake up, and the store buffer is repartitioned again.

Operating System Countermeasures. For operating systems that deploy kernel-private page tables with KAISER [25], the Meltdown countermeasure, every context switch also serializes the instruction stream when writing to CR3. We noticed that this has the unintended side-effect of draining outstanding stores from the store buffer [38], thereby preventing the WTF attack variants presented in this work. However, we note that this does distinctly *not* suffice as a general countermeasure against store buffer leakage since Intel's analysis [41] describes that stale values may still be recovered from the store buffer until explicitly overwritten.

The necessary software countermeasure for CPUs without silicon-level WTF mitigations is, therefore, to explicitly overwrite the entire store buffer on every context switch between user and kernel. To support this functionality, Intel [41] has released a microcode update that modifies the semantics of the legacy VERW instruction to overwrite (amongst others) the store buffer contents. Operating system kernels are required to execute a VERW dummy instruction (or equivalent legacy software code snippet [41]) upon every context switch to eliminate the possibility of reading stale kernel stores from user space.

Finally, we note that the above VERW countermeasure might not prevent attacks based on Store-to-Leak. To the best of our knowledge, no countermeasure has been suggested against the Store-to-Leak attack variants presented in this paper.

Gadget Finding. While Speculative Fetch+Bounce improves the usability of Spectre V1 gadgets, when attacking the kernel, we did not find such gadgets in kernel code. We will leave finding ways for detection gadgets in real-world applications for future work.

10 CONCLUSION

With the WTF shortcut, we demonstrate a novel Meltdown-type effect exploiting a previously unexplored microarchitectural component, namely the store buffer. The attack enables an unprivileged attacker to leak recently written values from the operating system. While WTF affects various processor generations, we showed that also recently introduced hardware mitigations are not sufficient and further mitigations need to be deployed.

We also show a way to leak the TLB state using the store buffer. We showed how to break KASLR on fully patched machines in 42 µs, as well as recover address space information from JavaScript. Next, we found that the Store-to-Leak TLB side channel facilitates the exploitation of Spectre gadgets. Finally, our work shows that the hardware fixes for Meltdown in recent CPUs are not sufficient.

ACKNOWLEDGMENTS

We want to thank the reviewers for their feedback, as well as Vedad Hadžić from Graz University of Technology and Julian Stecklina from Cyberus Technology for contributing ideas and experiments.

This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the Province of Styria and the Business Promotion Agencies of Styria and Carinthia. It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. It has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402), by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-19-C-0531, and by the National Science Foundation under grant CNS-1814406. Additional funding was provided by a generous gift from Intel and AMD.

The research presented in this paper was partially supported by the Research Fund KU Leuven. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO).

Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

A MEASURING THE STORE BUFFER SIZE

We now turn our attention to measuring the size of the store buffer. Intel advertises that Skylake processors have 56 entries in the store buffer [55]. We could not find any publications specifying the size of the store buffer in newer processors, but as both Kaby Lake and Coffee Lake R are not major architectures, we assume that the size of the store buffers has not changed. As a final experiment in this section, we now attempt to use the WTF shortcut to confirm this assumption. To that aim, we perform a sequence of store operations, each to a different address. We then use a faulty load aiming to trigger a WTF shortcut and retrieve the value stored in the first (oldest) store instruction. For each number of stores, we attempt 100 times at each of the 4096 page offsets, to a total of 409 600 per number of stores. Figure 12 shows the likelihood of triggering the WTF shortcut as a function of the number of stores for each of the processor and configurations we tried. We see that we can trigger the WTF shortcut provided that the sequence has up to 55 stores. This number matches the known data for Skylake and confirms our assumption that it has not changed in the newer processors.

The figure further shows that merely enabling hyperthreading does not change the store buffer capacity available to the process. However, running code on the second hyperthread of a core halves the available capacity, even if the code does not perform any store. This confirms that the store buffers are statically partitioned between the hyperthreads [42], and also shows that partitioning takes effect only when both hyperthreads are active.

REFERENCES

[1] 2018. Spectre Variant 4. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[2] 2019. https://bugs.chromium.org/p/chromium/issues/detail?id=821270

EXHIBIT 3
Page 14 of 16

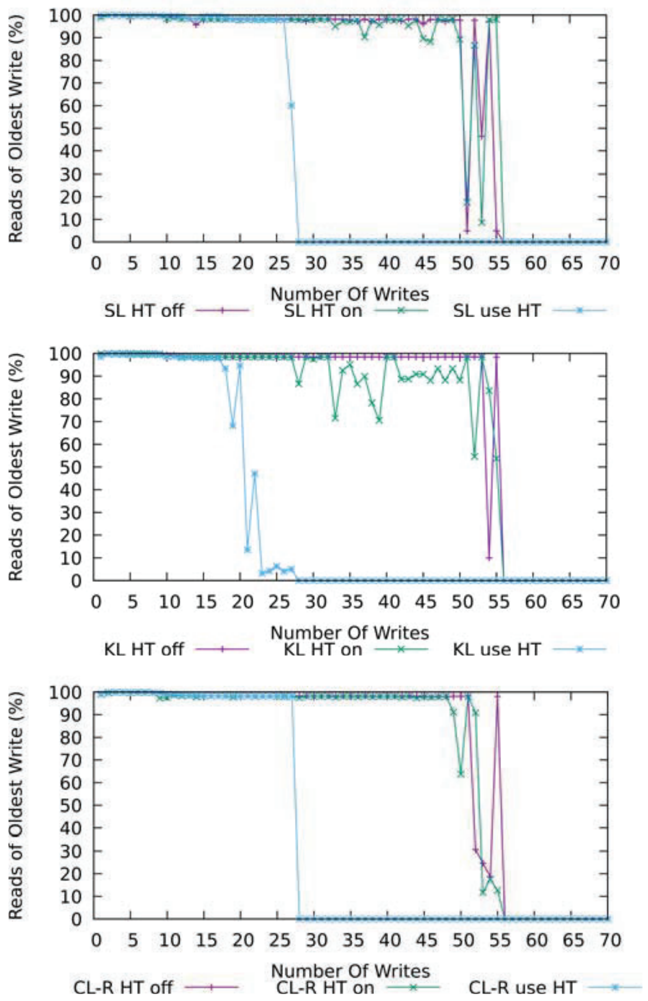


Figure 12: Measuring the size of the store buffer on Kaby Lake and Coffee Lake machines. In the experiment, we perform multiple writes to the store buffer and subsequently measure the probability of retrieving the value of the first (oldest) store. The results agree with 56 entries in the store buffer and with a static partitioning between hyperthreads.

[3] Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. 2002. Method and apparatus for performing a store operation. US Patent 6,378,062.

[4] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. 1998. Method and Apparatus for Dispatching and Executing a Load Operation to Memory. US Patent 5,717,882.

[5] ARM Limited. 2018. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism.

[6] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. 2014. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In *CHES*.

[7] Daniel J. Bernstein. 2004. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[8] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. 2017. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *CHES*. 555–576.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*.

[12] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/

[13] Jonathan Corbet. 2018. Meltdown strikes back: the L1 terminal fault vulnerability. https://lwn.net/Articles/762570/

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. (2016).

[15] Ian Cutress. 2018. Analyzing Core i9-9900K Performance with Spectre and Meltdown Hardware Mitigations. https://www.anandtech.com/show/13659/analyzing-core-i9-9900k-performance-with-spectre-and-meltdown-hardware-mitigations

[16] Ian Cutress. 2018. Spectre and Meltdown in Hardware: Intel Clarifies Whiskey Lake and Amber Lake. https://www.anandtech.com/show/13301/spectre-and-meltdown-in-hardware-intel-clarifies-whiskey-lake-and-amber-lake

[17] Vladimir Davydov. 2015. Idle memory tracking. https://lwn.net/Articles/643578/

[18] Agner Fog. 2016. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.

[19] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *CCS*. 845–858.

[20] Andy Glew, Glenn Hinton, and Haitham Akkary. 1997. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions. US Patent 5,680,565.

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*.

[22] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.

[23] Brendan Gregg. 2018. KPTI/KAISER Meltdown Initial Performance Regressions.

[24] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. 2019. Page Cache Attacks. In *CCS*.

[25] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.

[26] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *S&P*.

[27] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.

[28] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.

[30] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*.

[31] Shay Gueron. 2012. Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01.

[32] Michael Austin Halcrow. 2005. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Linux Symposium*.

[33] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. 2009. Resolving False Dependencies of Speculative Load Instructions. US Patent 7.603,527.

[34] Rodney E Hooker and Colin Eddy. 2013. Store-to-load forwarding based on load/store address computation source information comparisons. US Patent 8,533,438.

[35] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass.

[36] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.

[37] Intel. [n.d.]. Side Channel Mitigation by Product CPU Model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html

[38] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide.

[39] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf

[40] Intel. 2018. Speculative Execution Side Channel Mitigations. Revision 3.0.

[41] Intel. 2019. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling

[42] Intel. 2019. Intel 64 and IA-32 Architectures Optimization Reference Manual.

[43] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14*.

[44] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*.

[45] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*.

[46] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[47] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[48] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.

[49] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*.

[50] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[52] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[53] LWN. 2017. The current state of kernel page-table isolation. https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/

[54] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.

[55] Julius Mandelblat. [n.d.]. Technology Insight: Intel's Next Generation Microarchitecture Code Name Skylake. In *Intel Developer Forum (IDF15)*. https://en.wikichip.org/w/images/8/8f/Technology_Insight_Intel%E2%80%99s_Next_Generation_Microarchitecture_Code_Name_Skylake.pdf

[56] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.

[57] Microsoft. 2019. Lock your Windows 10 PC automatically when you step away from it. https://support.microsoft.com/en-us/help/4028111/windows-lock-your-windows-10-pc-automatically-when-you-step-away-from

[58] John Monaco. 2018. SoK: Keylogging Side Channels. In *S&P*.

[59] NIST. 2001. FIPS 197, Advanced Encryption Standard (AES).

[60] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. 2018. Spectre attack against SGX enclave.

[61] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*.

[62] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.

[63] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.

[64] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. 2017. To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In *CCS*. 1843–1855.

[65] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*.

[66] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.

[67] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[68] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[69] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.

[70] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*.

[71] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.

[72] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480* (2018).

[73] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv:1802.03802* (2018).

[74] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[75] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution*.

[76] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[77] Pepe Vila, Boris Köpf, and Jose Morales. 2019. Theory and Practice of Finding Eviction Sets. In *S&P*.

[78] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. [n.d.]. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf

[79] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*.

[80] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[81] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*.